INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 409

ÉVALUATION RÉCURSIVE DES GRAMMAIRES ATTRIBUÉES: DEUX IMPLANTATIONS

Martin JOURDAN

Mai 1985

EVALUATION RECURSIVE DES GRAMMAIRES ATTRIBUEES DEUX IMPLANTATIONS

Martin JOURDAN

INRIA
Domaine de Voluceau
BP 105
78150 - LE CHESNAY CEDEX
FRANCE

Résumé : Nous présentons deux algorithmes pour construire deux classes différentes d'évaluateurs récursifs pour les grammaires attribuées. Ces deux classes implantent une évaluation par nécessité dynamique, réduisant ainsi le nombre d'instances d'attribut à calculer. Nous présentons aussi un système réel fondé sur ces algorithmes et nous discutons son utilisation et ses performances.

Abstract : We present two algorithms to build two different classes of recursive evaluators for attribute grammars. Both those classes implement a dynamic evaluation by need, thus reducing the number of attributes instances to compute. We also present a practical system based on those algorithms and discuss its use and performances.

Mots-clés : grammaires attribuées - évaluation - circularité - Lisp - gestion de la mémoire - compilation.

Keywords : attribute grammars - evaluation - circularity - Lisp - memory management - compiling.

EVALUATION RECURSIVE DES GRAMMAIRES ATTRIBUEES: DEUX IMPLANTATIONS

Martin JOURDAN

INRIA
Domaine de Voluceau
BP 105
78153 - LE CHESNAY CEDEX
FRANCE

Résumé : Nous présentons deux algorithmes pour construire deux classes différentes d'évaluateurs récursifs pour les grammaires attribuées. Ces deux classes implantent une évaluation par nécessité dynamique, réduisant ainsi le nombre d'instances d'attribut à calculer. Nous présentons aussi un système réel fondé sur ces algorithmes et nous discutons son utilisation et ses performances.

I) Introduction

Les grammaires attribuées (GAs) sont un outil très agréable pour décrire des vérificateurs sémantiques pour les langages de programmation, des traducteurs, des compilateurs et, plus généralement, tout calcul dirigé par la syntaxe. Ceci est dû à leur caractère local, déclaratif, et à leur simplicité. Cependant ces caractéristiques, qui rendent les GAs si attractives pour décrire un calcul, rendent l'"exécution" de ce calcul problématique : l'évaluation d'attributs est non-déterministe dans le cas général.

Depuis l'article original de Knuth [Knu 68], de nombreux auteurs ont conçu des méthodes d'évaluation tendant à résoudre ces problèmes d'exécution. Nous pouvons classer grossièrement ces méthodes en deux groupes :

- celles qui utilisent un ordre d'évaluation statique, calculable à la construction; elles produisent des évaluateurs efficaces, mais au prix de restrictions généralement importantes sur la classe de GAs qu'elles acceptent: purement synthétisées [LRS 74], 1L - attribuées [Boc 76], kL - attribuées et simples multi-passes [RU 81], 1-visite et pures multi-passes [EF 81 a, b], ordonnées [Kas 80] et fortement (ou absolument) non-circulaires [KW 76, Kat 84, CF 82, Jou 84 a, c, d];

- celles qui acceptent n'importe quelle GA (non-circulaire), mais qui doivent déterminer à l'exécution un ordre d'évaluation dynamique; celles-ci sont généralement peu efficaces [Fan 72, Lor 77, CH 79, KR 79, Mad 80, JG 83].

Nous présentons l'implantation de deux méthodes d'évaluation efficaces. Toutes deux sont applicatives, au sens de [Eng 84]. La première, qui pourrait être classée dans le premier groupe, est applicable aux GAs fortement non-circulaires (FNC), et constitue une amélioration de la méthode de [CF 82]. Rappelons que la classe FNC est une très large sous-classe des GAs générales, comprenant toutes les GAs réelles rencontrées dans le domaine de la compilation. La seconde méthode se situe dans le deuxième groupe, mais est très efficace. Elle est optimale en temps (théoriquement), et détecte les circularités à l'exécution. Ces deux méthodes ont beaucoup d'aspects communs, donc les classer dans l'un ou l'autre groupe est assez artificiel.

Les deux méthodes implantent une évaluation par nécessité dynamique. Ceci signifie que, par exemple, si l'on a une règle sémantique conditionnelle de la forme :

$$a(X) = \underline{si}\ p\ \underline{alors}\ b(Y)\ \underline{sinon}\ c(Z)$$

et si a(X) doit par hasard être calculé, alors, selon la valeur de p, seule l'une des instances d'attributs b(Y) ou c(Z) sera calculée.

Les évaluateurs construits par nos deux méthodes sont des ensembles de fonctions mutuellement récursives, qui incluent le corps des règles sémantiques. Nous donnons les algorithmes qui permettent de construire ces évaluateurs à partir d'une description de GA.

Nous présentons un système pratique basé sur ces algorithmes. Le langage des règles sémantiques doit être un langage fonctionnel pour faciliter l'implantation des algorithmes. Nous avons choisi Lisp. Nous discutons ses avantages, en particulier au niveau de la gestion de la mémoire.

Nous présentons les caractéristiques générales de notre système, dont certaines se sont montrées très utiles dans le développement de GAs importantes. Nous présentons enfin quelques applications de notre système et un aperçu de ses performances.

II) Notations

Grammaire context-free $G = (N, T, P, Z)$, où N est l'ensemble des non-terminaux, T l'ensemble des terminaux (inutilisé dans la suite), P l'ensemble des productions et $Z \in N$ l'axiome.

Productions : $p : X_0 \rightarrow X_1 \; X_2 \ldots X_{n_p}$, $X_i \in N$, $p \in P$.
Attributs : a, I(X) et S(X) attributs hérités et synthétisés de $X \in N$, $A(X) = I(X) \cup S(X)$, $I = \bigcup_{X \in N} I(X)$, $S = \bigcup_{X \in N} S(X)$, $A = I \cup S$. On suppose $I(X) \cap S(Y) = \emptyset$, $\forall X,Y \in N$
Occurrences d'attributs : a(X)
Règles sémantiques : $a(X_k) = f_{p,a,k}(b_1(X_{k1}), \ldots . b_m(X_{k_m}))$.

Nous n'exigeons pas que la GA soit en forme normale.

Les arbres de dérivation (arbres syntaxiques) et l'application d'une production p à un noeud u d'un arbre t sont définis de manière habituelle.

Nous définissons la valeur sémantique d'un arbre t comme étant la liste des valeurs des attributs (synthétisés) de la racine de t. Nous restreignons le problème de l'évaluation des attributs au calcul de cette valeur sémantique, comme le fait Knuth [Knu 68], et non pas au calcul de tous les attributs.

III) Première méthode : FNC

Cette méthode est dérivée de la construction de Courcelle et Franchi-Zannettacci pour les GAs FNC [CF 82]. Elle a été améliorée par l'auteur [Jou82, Jou84 a,c,d]. Elle repose sur le fait que la valeur d'un attribut synthétisé à un noeud d'un arbre ne dépend que :

- du sous-arbre issu de ce noeud,
- des valeurs des attributs hérités à ce noeud.

Il est donc tout-à-fait naturel d'associer à chaque attribut synthétisé une fonction, prenant comme arguments un (sous-) arbre et les valeurs de certains attributs hérités - ceux qui sont nécessaires - , et rendant la valeur de cet attribut, pour cet arbre et ces valeurs.

Les attributs hérités passés en argument sont choisis par le sélecteur d'argument, ASel. ASel(a,X), où $X \in N$ et $a \in S(X)$, est l'ensemble des attributs hérités de X dont a peut dépendre pour n'importe quel arbre issus de X . Le calcul du sélecteur d'arguments se fait par un algorithme (Fig.1) [CF 82] ressemblant au "faux" algorithme de Knuth [Knu 68], en un temps polynômial. Il peut être grandement amélioré [DJL 84]. Le calcul du sélecteur d'argument échoue si la GA n'est pas FNC; comme dit précédemment, ceci n'est pas une trop forte contrainte.

La première amélioration apportée à la méthode originelle a été de faire en sorte que les fonctions constituant l'évaluateur décorent l'arbre avec les valeurs qu'elles rendent. Ainsi la valeur de chaque attribut synthétisé n'est calculée qu'au plus une fois, au premier appel, et ensuite elle est simplement récupérée de l'arbre. Ceci réduit le temps de calcul - la méthode originelle est de complexité exponentielle - de façon très importante dès que la GA est quelque peu importante [Jou 82].

Le sélecteur d'arguments est fonction d'un attribut synthétisé et d'un non-terminal; nous devrions donc produire une fonction pour chaque couple. Cependant nous avons décidé de regrouper toutes les fonctions associées au même attribut. Ceci a deux effets :

- réduction de la taille du code engendré, par partage des parties communes de ces fonctions (le "bookkeeping");
- réduction de la taille des arbres, par élimination des productions simples, c'est-à-dire des productions de la forme $X \longrightarrow Y$ dont la sémantique est réduite à des règles de copie.

Pour résoudre le problème de la compatibilité des listes d'arguments (il est possible que ASel $(a,X) \neq$ ASel (a,Y) pour $X \neq Y$), ces fonctions "comprimées" n'auront que deux arguments : le sous-arbre et un aggrégat des valeurs des attributs hérités. Chaque règle sémantique dans le corps de la fonction extraira de cet aggrégat les valeurs nécessaires.

Pour cet algorithme, un sélecteur d'arguments $\gamma : SxN \longrightarrow \mathcal{P}(I)$ sera considéré comme un ensemble de triplets $R(\gamma) \subset IxSxN$ tel que

$$\langle y, a, X\rangle \in R(\gamma) \Longleftrightarrow y \in \gamma(a, X)$$

et pour tout tel ensemble de triplets R, le sélecteur d'arguments correspondant sera noté $\Gamma(R)$.

```
1: i := 0; R := ∅;

2: répéter
       convergence := vrai; i := i + 1;
       pour tout X ∈ N faire
           pour tout p ∈ P, LHS(p) = X faire
               calculer -p,Γ(R)+>;
               pour tout a ∈ S(X) faire
                   pour tout y ∈ I(X) faire
                       si a(X) -p,Γ(R)+> y(X) et <y, a, X> ∉ R alors
                           R := R ∪ {<y, a, X>};
                           convergence := faux
                       fin si
                   fin pour
               fin pour
           fin pour
       fin pour
   jusqu'à convergence;

3: ASel := Γ(R); /* ASel est le sélecteur d'arguments clos minimal */
```

Figure 1

La structure de chaque fonction est présentée en figure 2. Elle inclut d'autres améliorations décrites dans la section V. La figure 3 présente l'algorithme produisant, à partir de la description d'une GA, l'ensemble des fonctions constituant l'évaluateur.

Le rôle du paramètre booléen "expand ?" est de contrôler la réécriture automatique de la GA en forme normale. Si nous voulons que cette réécriture ait lieu systématiquement, ce paramètre doit être toujours vrai. Ici il est faux pour les occurrences d'attributs synthétisés du non-terminal en partie gauche de la production qui apparaissent dans le corps des règles sémantiques de cette production (ce qui signifie que la GA n'est pas en forme normale); ainsi on produit l'appel à la fonction correspondante, ce qui a deux effets [Jou 84 d]:

- réduction de la taille des fonctions,
- utilisation du fait que la valeur rendue par ces appels est stockée.

L'algorithme de la figure 3 construit aussi une fonction "valeur-sémantique". Pour calculer la valeur sémantique d'un texte source, il suffit d'analyser le texte pour produire son arbre syntaxique, et d'appeler la fonction "valeur-sémantique" en lui passant cet arbre en argument.

La caractéristique la plus intéressante de cette méthode est l'évaluation par nécessité dynamique : seules les instances d'attributs nécessaires pour calculer la valeur sémantique sont évaluées. De plus cette nécessité est dynamique : si une règle sémantique est une conditionnelle de la forme :

$$a(X) = \underline{si}\ \ p\ \ \underline{alors}\ \ b(Y)\ \ \underline{sinon}\ \ c(Z)$$

et si $a(X)$ doit en fait être évalué, alors, selon la valeur (dynamique) de p, seule l'une des deux instances $b(Y)$ ou $c(Z)$ sera calculée, avec les instances dont elle dépend. Virtuellement le graphe de dépendance est modifié par le flot de contrôle des règles sémantiques, comme résumé en figure 4. Bien évidemment, le calcul du sélecteur d'argument prend en compte les dépendances statiques.

Un (petit) inconvénient de cette méthode est que les valeurs des attributs hérités qui apparaissent dans la liste d'arguments à l'appel d'une fonction sont recalculées à chaque appel, puisqu'elles ne sont pas stockées, et ceci même si la valeur de l'attribut synthétisé rendue par cet appel n'est pas recalculée mais simplement extraite de l'arbre.

```
fonction ψa (arbre, parm-liste);
/* label (arbre) = p ∈ P, p: X0 --> X1 .. Xnp , a ∈ S (X0) */
   si racine (arbre) contient déjà une valeur pour a alors
       récupérer et rendre cette valeur
   sinon cas label (arbre) est
            :
            :
       p: ψa  := fp,a,0   (
                   :
                 récupérer (yi , parm-liste),
                 /* pour les occurrences yi(X0), yi ∈ I */
                    :
                 ψbj  (arbre, parm-liste),
                 /* pour les occurrences bj(X0), bj ∈ S:
                    forme non-normale */
                    :
                 ψcj  (fils (arbre, kj),
                       liste (fp,z1,kj (...),
                                  :
                              fp,zm',kj (...))),
                 /* pour les occurrences cj (Xkj ) avec 1 ≤ kj ≤ np,
                    cj ∈ S(Xkj )et ASel (cj, Xkj )= {z1, .., zm'} */
                 /* les autres occurrences sont réécrites en forme
                    normale en étendant leur définition */
                    :
                    );
            :
            :
       fin cas;
       stocker ψa dans racine (arbre);
       rendre ψa
   fin si
fin ψa;
```

Figure 2

```
procédure print-attr-def (a, p, k, expand?);
/* écrit la "définition" de l'attribut "a" du symbole à la
   position "k" (0 pour la partie gauche, 1, 2, .., n_p pour la
   partie droite) dans la production "p" */
/* Le paramètre booléen "expand?" précise s'il faut réécrire
   cette définition en forme normale ou non */

    si a ∈ I et k = 0 alors
        /* attribut hérité du non-terminal en partie gauche:
           fait partie de l'aggrégat de valeur en paramètre */
        écrire ("récupérer (a, parm-liste)")

    sinsi a ∈ S et k > 0 alors
        /* attribut synthétisé d'un non-terminal en partie
           droite: il faut engendrer l'appel à la fonction
           correspondante en construisant l'aggrégat */
        écrire (" ψ_a (fils (arbre, k), liste (");
        pour tout y ∈ ASel (a, X_k) faire
            écrire ("paire (y, ");
            print-attr-def (y, p, k, faux)
            écrire ("),");
            /* on ne s'occupe pas des virgules en trop */
        fin pour;
        écrire ("))")

    sinsi a ∈ S et non expand? alors
        écrire (" ψ_a (arbre, parm-liste)")

    sinon /* il y a une règle sémantique: f_p,a,k  */
        écrire le membre droit de la règle sémantique corres-
            pondant à a, p, k, en remplaçant chaque
            occurrence d'attribut b_j(X_kj) par
            print-attr-def (b_j, p, k_j, faux)

    fin si
fin print-attr-def;
```

```
/* corps de l'algorithme */
pour tout a ∈ S faire
    écrire ("fonction ψ_a (arbre, parm-liste);
            si racine (arbre) contient deja une valeur pour a
            alors recuperer et rendre cette valeur
            sinon cas label (arbre) est");
    pour tout p ∈ P, p non simple faire
        si a ∈ S(LHS(p)) alors
            écrire ("p: ψ_a := ");
            print-attr-def (a, p, 0, vrai);
            /* il faut effectivement écrire cette définition */
            écrire (";")
        fin si
    fin pour;
    écrire ("fin cas;
            stocker ψ_a dans racine (arbre);
            rendre  ψ_a
            fin si
            fin ψ_a ;")
fin pour;

/* valeur sémantique */
écrire ("fonction valeur-sémantique (arbre);");
écrire ("valeur-sémantique := liste (");
pour tout a ∈ S(Z) /* axiome */ faire
    écrire ("ψ_a (arbre, liste-vide), ")
fin pour;
écrire (")");
écrire ("fin valeur-sémantique;");

stop.
```

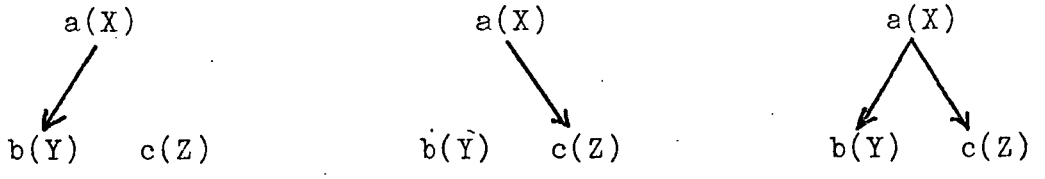
Figure 3

Cependant ceci n'est pas un gros problème puisque dans les GAs pratiques les attributs hérités dépendent "simplement" d'attributs synthétisés qui, eux, sont stockés [Kav 84].

Notons que dans cette méthode d'évaluation la hauteur de la pile des appels de fonctions est proportionnelle à la hauteur de l'arbre.

La méthode FNC se rapproche des travaux de Kennedy et Warren [KW 76], Saarinen [Saa 78] et surtout de Katamaya [Kat 84]. En particulier la classe FNC est exactement la classe des GAs absolument non-circulaires. Cependant la méthode FNC a des bases théoriques beaucoup plus solides [CF 82] et est beaucoup plus simple à mettre en oeuvre.

a(X) = si p alors b(Y) sinon c(Z)



graphe de dépendance quand p = vrai

graphe de dépendance quand p = faux

graphe de dépendance "statique"

Figure 4

IV) Seconde méthode : ERN

ERN signifie "Evaluation Récursive par Nécessité". Dans cette méthode nous produisons une fonction pour chaque attribut, synthétisé ou hérité. Ces fonctions ont la même structure globale que pour FNC - une instruction "cas" qui sélectionne la bonne règle sémantique - mais pour ERN toutes les fonctions n'ont qu'un seul argument, un sous-arbre; pour obtenir la valeur d'une instance d'attribut apparaissant dans le membre droit d'une règle sémantique, on appelle simplement la fonction correspondante.

Comme pour FNC, ces fonctions rendent la valeur de l'attribut correspondant à la racine de l'arbre passé en argument, après l'avoir soit calculée et stockée soit simplement extraite de l'arbre. Pour pouvoir calculer les attributs hérités, chaque noeud de l'arbre contient un pointeur vers son père et son index dans la partie droite de la production appliquée au père. La structure des fonctions est décrite en figures 5 (attributs synthétisés) et 6 (attributs hérités).

Cette construction est applicable à tous les GAs. De plus il est facile de détecter les vraies circularités à l'exécution, à l'aide d'un simple marqueur. Cette idée est empruntée à Jalili et Gallier [JG 83].

En partant d'une description de GA, l'ensemble des fonctions constituant l'évaluateur est construit par un algorithme très similaire à celui de FNC (voir [Jou 84 b,d]).Les productions simples sont traitées de la même façon. Comme pour FNC, le calcul de la valeur sémantique d'un arbre se réduit à l'appel successif des fonctions correspondant aux attributs de la racine, auxquelles on passe cet arbre en argument.

Au cours de l'évaluation, la pile des appels de fonction simule un parcours haut-bas du graphe de dépendance complet - qui n'est pas construit - sauf que, à cause du stockage des valeurs, au retour de la visite de chaque noeud, ce noeud devient une feuille, et que la relation de dépendance est dynamiquement modifiée par le flot de contrôle des règles sémantiques (figure 4). Il est donc facile de voir que notre méthode est dynamiquement optimale : nous ne calculons que les instances d'attributs strictement nécessaires pour obtenir la valeur sémantique de l'arbre.

```
fonction ψa (arbre);
    si racine (arbre) contient déjà une valeur pour a alors
        récupérer et rendre cette valeur
    sinsi racine (arbre) contient une marque pour a alors
        écrire ("Circularite...");
        stop
    sinon marquer a dans racine (arbre);
        cas label (arbre) est
                :
                :
            p: ψa := fp,a,0 (
                            :
                        ψbj (fils (arbre, k )),
                       /* j pour toutes les occurrences bj(Xkj)
                          en posant fils (arbre, 0) = arbre */
                            :
                            );
                :
                :
        fin cas;
        stocker ψa dans racine (arbre);
        rendre ψa
    fin si
fin ψa ;
```

Figure 5

```
fonction ψa (arbre);
   si racine (arbre) contient déjà une valeur pour a alors
      récupérer et rendre cette valeur
   Sinsi racine (arbre) contient une marque pour a alors
         écrire ("Circularite...");
         stop
     sinon marquer a dans racine (arbre);
         soit (pos := indice(arbre),
               papa := père (arbre)) dans
            soit arbre := papa dans
                cas [label (arbre), pos] est
                /* [] est l'opérateur de liste */
                        :
                        :
                    [p, k]: ψa := fp,a,k  (
                                    :
                                    ψbj(fils (arbre, kj)),
                            /* pour toutes les occurrences bj(Xkj)
                               en posant fils (arbre, 0) = arbre */
                                    :
                                    );
                        :
                        :
                fin cas
            fin soit
         fin soit;
         stocker ψa dans racine (arbre);
         rendre ψa
    fin si ψa
fin ψa ;
```

Figure 6

Dans la méthode ERN, la hauteur de la pile des appels de fonction peut être aussi grande que la taille de l'arbre : ceci peut être très grand, et interdire l'analyse de textes sources longs. Cependant, comme ERN ne nécessite pas de coûteux tests de non-circularité, elle est bien adaptée au développement d'une nouvelle GA : le temps de construction après chaque modification est beaucoup plus réduit que pour FNC.

V) Une implantation en Lisp et ses optimisations

Nous avons réalisé un système, nommé FNC/ERN, qui implante les deux méthodes précédentes. Actuellement il tourne sur Multics; sa portabilité est faible car FNC/ERN est lié au système SYNTAX (marque déposée de l'INRIA), un générateur d'analyseurs lexico-syntaxiques doué de grandes qualités mais peu portable.

L'implantation de nos algorithmes est plus facile, c'est-à-dire plus proche de la théorie, si les règles sémantiques sont écrites dans un langage fonctionnel (applicatif) plutôt que dans un langage impératif. Cependant ceci n'est pas une condition nécessaire : tout langage possédant des fonctions, par exemple Pascal, PL/1 ou Ada, est applicable. Mais pour de nombreuses raisons, nous avons choisi Lisp. En voici quelques-unes :

- raisons historiques : FNC a été conçu comme une partie intégrante de Perluette [Des 82] qui produit des compilateurs écrits en Lisp;

- pas de typage fort : ceci nous permet une plus grande souplesse et une plus grande régularité dans l'implantation des fonctions et des arbres;

- Lisp est un langage bien diffusé dans le monde entier, bénéficiant d'implantations efficaces (Maclisp, Lelisp,..);

- gestion efficace et automatique de la mémoire.

La gestion de la mémoire est un des problèmes les plus gênants pour une diffusion importante de la méthode des GAs : quand le texte source est un tant soit peu long, la taille de l'arbre et le nombre d'instances d'attributs à calculer et à stocker devient très important. Ceci peut interdir tout usage des GAs, surtout sur des micro ou mini-ordinateurs. Lisp propose une solution élégante à ce problème.

Le premier point est que Lisp manipule ses données à travers des pointeurs; donc copier une donnée revient à copier un pointeur. Dans les GAs usuelles, les règles de copie peuvent représenter jusqu'à 60% du nombre de règles sémantiques; si les valeurs des attributs occupent une place mémoire importante, beaucoup de place peut être économisée par l'emploi de pointeurs. Le deuxième point est que la plupart des structures de données composites peuvent être partagées entre plusieurs variables si elles sont implantées par des listes. Par exemple, dans une règle comme :

a(X) = (cons anything b(Y))

la place mémoire utilisée pour stocker à la fois a(X), b(Y) et "anything" ne représente qu'une seule cellule de liste en plus de celle utilisée pour stocker b(Y) et "anything". Ainsi les structures de données importantes comme les tables de symboles ne doivent plus être dupliquées et peuvent être partagées (en partie ou en totalité) entre plusieurs instances d'attributs.

Cette gestion de la mémoire s'apparente un peu à celle de Räihä [Räi 79], mais est effectuée automatiquement par Lisp, de façon dynamique. Nous n'avons pas la possibilité de désallocation de Räihä, mais celle-ci n'est applicable que pour les classes de GAs telles qu'il est possible de déterminer à la construction le moment où une instance d'attribut devient inutile. Ceci n'est pas possible pour les GAs FNC ou générales, surtout compte tenu de notre évaluation par nécessité dynamique.

Cependant nous avons implanté quelques optimisations qui permettent d'économiser de la place mémoire :

- en comptant dynamiquement le nombre d'attributs synthétisés évalués à chaque noeud, on peut savoir si un sous-arbre est complètement évalué; on peut alors le détruire complètement (sauf la racine); la place mémoire sera récupérée automatiquement par le ramasse-miettes (garbage-collector) de Lisp;

- nous avons déterminé des conditions statiques suffisantes pour que toutes les instances d'un attribut donné ne soient utilisées qu'une seule fois; quand cela est le cas, ces instances ne sont pas stockées dans l'arbre : elles n'existent que comme valeurs de retour de fonction;

- enfin, pour FNC, la gestion de l'aggrégat des valeurs des attributs hérités passé en argument est conçue de façon à minimiser la taille occupée par cet aggrégat [Jou 84 d].

Le système FNC/ERN est composé d'un constructeur et d'un environnement d'exécution. Le constructeur lit une description de GA, la vérifie et produit ensuite l'évaluation correspondant en utilisant l'une des deux méthodes selon le choix de l'utilisateur. Cet évaluateur, c'est-à-dire l'ensemble des fonctions Lisp, peut être éventuellement compilé, si le système Lisp employé le permet.

L'environnement d'exécution est un petit sous-système Lisp dont le rôle est de charger l'évaluateur, appeler les analyseurs lexico-syntaxiques pour construire l'arbre de dérivation, et ensuite appeler la fonction "valeur-sémantique" pour calculer les attributs.

Le constructeur est écrit en PL/1 (environ 4000 lignes) et utilise des outils de haut niveau comme le système SYNTAX. Le test de non-circularité forte et la construction du sélecteur d'argument sont optimisés par les méthodes de [DJL 84].

De nombreuses caractéristiques rendent le système facile à utiliser; citons des attributs, des variables et des fonctions prédéfinis, des messages d'erreurs explicites, un rattrapage d'erreurs puissant dû au système SYNTAX et accessible au niveau des attributs, des tables de références croisées extensives facilitant l'analyse d'une GA (sur demande), et bientôt un système interactif de trace des circularités dans une GA. Pour plus de détails voir [Jou 84 d].

La caractéristique la plus utile de FNC/ERN est sa capacité de fabriquer automatiquement les déclarations d'attributs et les règles sémantiques manquantes. L'expérience prouve que l'on peut gagner ainsi jusqu'à 50% du nombre de lignes à écrire pour décrire une GA.

VI) Applications et performances

Le système FNC/ERN est utilisé à l'INRIA et sur d'autres sites Multics depuis maintenant trois ans et donne tout-à-fait satisfaction. Ses applications actuelles concernent essentiellement la compilation. Un certain nombre de langages-jouets ont été implantés. FNC/ERN est intégré au système Perluette [Des 82], à la fois comme la première phase des compilateurs produits et comme traducteur pour certaines des entrées de Perluette.

Nous avons utilisé FNC/ERN pour développer des applications de taille réaliste, dont les deux plus intéressantes sont un compilateur complet (ISO) de Pascal en P-code, et un traducteur de Pascal en Ada [BDJ 83]. Ces GAs ont été développées et testées en respectivement trois et cinq hommes-mois; ceci est une preuve de l'efficacité de la méthode des attributs pour l'écriture de compilateurs. Mais la méthode est aussi efficace à l'exécution : le compilateur Pascal→P-code écrit avec FNC/ERN ne "tourne" que 2,5 fois moins rapidement qu'un compilateur écrit à la main qui n'est pas aussi puissant; et bien sûr, le premier possède un bien meilleur rattrapage d'erreurs, des messages d'erreur explicites,... et est beaucoup plus facile à mettre au point, à modifier et à maintenir. Ceci montre que les attributs sont maintenant arrivés à un niveau industriel.

Pour ce qui est de l'implantation pratique, l'expérience montre que, bien que ERN calcule moins d'attributs que FNC, elle est moins efficace en temps; ceci est dû au fait que le mécanisme des appels de fonction est beaucoup plus sollicité dans la première méthode. Cependant ERN est malgré tout utile : puisqu'elle ne nécessite pas de test de circularité, coûteux même quand il est optimisé, le temps de construction pour la même GA est beaucoup plus court avec FNC. ERN trouve donc sa place au cours du développement d'une nouvelle GA : on l'écrit et on la teste avec ERN, puis quand la GA est au point, on utilise FNC pour obtenir un évaluateur plus efficace.

VII) Conclusion

Nous avons présenté deux algorithmes pour construire des évaluateurs efficaces pour les GAs, ainsi qu'un système pratique qui les implante. Les évaluateurs sont des ensembles de fonctions mutuellement récursives, et implantent une évaluation par nécessité dynamique, ce qui diminue le nombre d'instances d'attributs à évaluer pour obtenir la valeur sémantique d'un texte.

Nous espérons avoir montré que l'utilisation des GAs dans des applications pratiques est maintenant possible. Leur facilité d'utilisation réduit le coût du développement des grands projets logiciels d'une manière drastique, et apporte une grande fiabilité et une facilité importante de modification et de maintenance.

VIII) Bibliographie

[BDJ 83] P.Boullier, Ph.Deschamp et M.Jourdan : "Spécification et Réalisation d'un Traducteur Pascal-Ada", Journées Ada de l'AFCET, Lausanne (J.André et A.Strohmeier eds.), pp 9-29 (Décembre 1983).

[Boc 76] G.V. Bochmann: "Semantic Evaluation from Left to Right", CACM 19,2, pp 55-62 (Février 1976).

[CF 82] B.Courcelle et P.Franchi-Zannettacci: "Attribute Grammars and Recursive Program Schemes" (part I and 2), Theoretical Computer Science 17,2 et 3, pp 163-191 et 235-257 (1982).

[CH 79] R.Cohen et E.Harry: "Automatic Generation of Near-Optimal Linear-Time Translators for Attribute Grammars, 6th ACM POPL, San Antonio, TX, pp 121-134 (Janvier 1979).

[DJL 84] P.Deransart, M.Jourdan et B.Lorho: "Speeding up Circularity Tests for Attribute Grammars", Acta Informatica 21, pp 375-391 (1984).

[Des 82] Ph.Deschamp: "Perluette: a Compilers Producing System using Abstract Data Types", 5th Int.Symp. on Programming, Torino, (M.Dezani-Ciancagliori et V.Montanari eds), LNCS 137, Springer-Verlag, pp 63-77 (Avril 1982).

[EF 81 a] J.Engelfriet et G.Filé: "The Formal Power of One-Visit Attribute Grammars", Acta Informatica 16,3,pp 275-302 (1981).

[EF 81 b] J.Engelfriet et G.Filé: "Passes and Paths of Attribute Grammars", Information and Control 49,2,pp 125-169 (Mai 1981).

[Eng 84] J.Engelfriet: "Attribute Grammars: Attribute Evaluation Methods", Methods and Tools for Compiler Construction (B.Lorho ed), Cours INRIA-CCE, Cambridge University Press, pp 103-138 (1984).

[Fan 72] I.Fang: "FOLDS, a Declarative Formal Language Definition System", Phd thesis, rapport STAN-CS-72-329, Computer Science Dept, Stanford University, Stanford, Ca (Décembre 1972).

[JG 83] F.Jalili et J.H.Gallier: "A General Incremental Evaluator for Attribute Grammars", Technical report, Dept. of Computer Science, Moore School of Electrical Engineering D2, University of Pennsylvania, Philadelphia, PA. (Mars 1983).

[Jou 82] M.Jourdan: "Un Evaluateur Efficace pour les Grammaires Attribuées Fortement Non-Circulaires", rapport 82-39, LITP, Paris (Sept.1982).

[Jou 84 a] M.Jourdan "Recursive Evaluators for Attribute Grammars: an Implementation", Methods and Tools for Compiler Construction (B.Lorho ed), Cours INRIA-CCE, Cambridge University Press, pp 139-164 (1984).

[Jou 84 b] M.Jourdan: "An Optimal-Time Recursive Evaluator for Attribute Grammars", 6th Int.Symp. on Programming, Toulouse (M.Paul et B. Robinet eds), LNCS 167, Springer-Verlag, pp 167-178 (Avril 1984).

[Jou 84 c] M.Jourdan: "Les Grammaires Attribuées: Implantation, Applications, Optimisations", Thèse de Docteur-Ingénieur, Université de Paris VII (Mai 1984). Aussi rapport 84-50, LITP, Paris.

[Jou 84 d] M.Jourdan: "Strongly Non-Circular Attribute Grammars and Their Recursive Evaluation", ACM SIGPLAN'84 Symp. on Compiler Construction, Montréal, SIGPLAN Notices 19,6, pp 81-93 (Juin 1984).

[Kas 80] U.Kastens: "Ordered Attribute Grammars", Acta Informatica 13,3, pp 229-256 (1980).

[Kat 84] T.Katamaya: "Translation of Attribute Grammars into Procedures", ACM TOPLAS 6,3, pp 345-369 (Juillet 1984).

[Kav 84] C.Kaviani: "Comparaison de deux Méthodes d'Evaluation d'Attributs Sémantiques: GAG et FNC", thèse de 3° cycle, Université de Paris VI (Janvier 1984).

[Knu 68] D.E.Knuth: "Semantics of Context-Free Languages", Mathematical Systems Theory 2,2, pp 127-145 (Juin 1968).

[KR 79] K.Kennedy et J.Ramanathan: "A Deterministic Attribute Grammar Evaluator based on Dynamic Sequencing", ACM TOPLAS 1,1, pp 142-160 (Juillet 1979).

[KW 76] K.Kennedy et S.K.Warren: "Automatic Generation of Efficient Evaluators for Attribute Grammars", 3rd ACM POPL, Atlanta, Ge, pp 32-49 (Janvier 1976).

[Lor 77] B.Lorho: "Semantic Attributes Processing in the System DELTA", Methods of Algorithmic Language Implantation (A.Ershov et C.H.A. Koster eds.),LNCS 47, Springer-Verlag, pp 21-40 (1977).

[LRS 74] P.M.Lewis, D.J.Rosenkrantz et R.E.Stearns: "Attributed Translations", JCSS 9,4, pp 279-307 (Décembre 1974).

[Mad 80] O.L.Madsen: "On Defining Semantics by Means of Extended Attribute Grammars", Semantics Directed Compiler Generation (N.D. Jones ed), LNCS 94, Springer-Verlag, pp 259-299 (1980).

[Räï 79] K.J.Räihä: "Dynamic Allocation of Space for Attributes Instances in Multi-Pass Evaluators of Attribute Grammars", ACM SIGPLAN'79 Symp. on Compiler Construction, Denver, Co, SIGPLAN Notices 14,8, pp 26-38 (Août 1979).

[Ru 81] K.J.Räihä: et E.Ukkonen: "Minimizing the Number of Evaluation Passes for Attribute Grammars", SIAM Journal on Computing 10,4, pp 772-786 (Novembre 1981).

[Saa 78] M.Saarinen: "On Constructhing Efficient Evaluators for Attribute Grammars", 5th ICALP, Udine (G.Ausiello et C.Böhm eds.), LNCS 62, Springer-Verlag, pp 382-397 (Juillet 1978).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique